10 End-to-end encryption

This chapter covers

- End-to-end encryption and its importance
- Different attempts at solving email encryption
- How end-to-end encryption is changing the landscape of messaging

Chapter 9 explained transport security via protocols like TLS and Noise. At the same time, I spent quite some time explaining where trust is rooted on the web: hundreds of certificate authorities (CAs) trusted by your browser and operating system. While not perfect, this system has worked so far for the web, which is a complex network of participants who know nothing of each other.

This problem of finding ways to trust others (and their public keys) and making it scale is at the center of real-world cryptography. A famous cryptographer was once heard saying, "Symmetric crypto is solved," to describe a field of research that had overstayed its welcome. And, for the most part, the statement was true. We seldom have issues encrypting communications, and we have strong confidence in the current encryption algorithms we use. When it comes to encryption, most engineering challenges are not about the algorithms themselves anymore, but about who Alice and Bob are and how to prove it.

Cryptography does not provide one solution to trust but many different ones that are more or less practical, depending on the context. In this chapter, I will survey some of the different techniques that people and applications use to create trust between users.

10.1 Why end-to-end encryption?

This chapter starts with a "why" instead of a "what." This is because end-to-end encryption is a concept more than a cryptographic protocol; it's a concept of securing communications between two (or more) participants across an adversarial path. I started this book with a simple example: Queen Alice wanted to send a message to Lord Bob without anyone in the middle being able to see it. Nowadays, many applications like email and messaging exist to connect users, and most of them seldom encrypt messages from soup to nuts.

You might ask, isn't TLS enough? In theory, it could be. You learned in chapter 9 that TLS is used in many places to secure communications. But end-to-end encryption is a concept that involves actual human beings. In contrast, TLS is most often used by systems that are "men-in-the-middle" by design (see figure 10.1). In these systems, TLS is only used to protect the communications between a central server and its users, allowing the server to see everything. Effectively, these MITM servers sit in between users, are necessary for the application to function, and are *trusted third parties* of the protocol. That is to say, we have to trust these parts of the system in order for the protocol to be considered secure (spoiler alert: that's not a great protocol).

In practice, much worse topologies exist. Communications between a user and a server can go through many network hops, and some of these hops might be machines

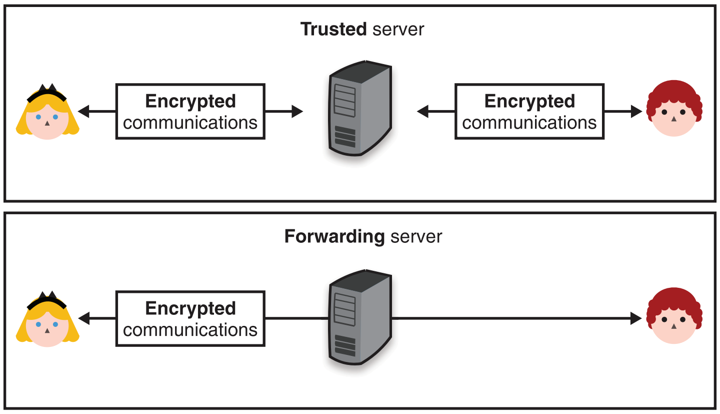


Figure 10.1 In most systems, a central server (top diagram) relays messages between users. A secure connection is usually established between a user and the central server, which can thus see all user messages. A protocol providing end-to-end encryption (bottom diagram) encrypts communications from one user up to its intended recipient, preventing any server in the middle from observing messages in cleartext.

that look at the traffic (often referred to as *middleboxes*). Even if traffic is encrypted, some middleboxes are set up to end the TLS connection (we say that they *terminate TLS*) and either forward the traffic in clear from that point on or start another TLS connection with the next hop. TLS termination is sometimes done for "good" reasons: to better filter traffic, balance connections geographically or within a data center, and so on. This adds to the attack surface as traffic is now visible in the clear in more places. Sometimes, TLS termination is done for "bad" reasons: to intercept, record, and spy on traffic.

In 2015, Lenovo was caught selling laptops with pre-installed custom CAs (covered in chapter 9) and software. The software was MITM'ing HTTPS connections using Lenovo's CAs and injecting ads into web pages. More concerning, large countries like China and Russia have been caught redirecting traffic on the internet, making it pass through their networks in order to intercept and observe connections. In 2013, Edward Snowden leaked a massive number of documents from NSA showing the abuses of many governments (not just the US) in spying on people's communications by intercepting the internet cables that link the world together.

Owning and seeing user data is also a liability for companies. As I've mentioned many times in this book, breaches and hacks happen way too often and can be devastating for the credibility of a company. From a legal standpoint, laws like the General Data Protection Regulation (GDPR) can end up costing organizations a lot of money. Government requests like the infamous National Security Letters (NSLs) that sometimes prevent companies and people involved from even mentioning that they have received the letters (so-called gag orders) can be seen as additional cost and stress to an organization, too, unless you have nothing much to share.

Bottom line, if you're using a popular online application, chances are that one or more governments already have access or have the ability to gain access to everything you wrote or uploaded there. Depending on an application's *threat model* (what the application wants to protect against) or the threat model of an application's most vulnerable users, end-to-end encryption plays a major role in ensuring confidentiality and privacy of end users.

This chapter covers different techniques and protocols that have been created in order to create trust between people. In particular, you will learn about how email encryption works today and how secure messaging is changing the landscape of end-to-end encrypted communications.

10.2 A root of trust nowhere to be found

One of the simplest scenarios for end-to-end encryption is the following: Alice wants to send an encrypted file to Bob over the internet. With all the cryptographic algorithms you learned about in the first chapters of this book, you can probably think of a way to do this. For example

1. Bob sends his public key to Alice.
2. Alice encrypts the file with Bob's public key and sends it to Bob.

Perhaps Alice and Bob can meet in real life or use another secure channel they already share to exchange the public key in the first message. If this is possible, we say that they have an *out-of-band* way of creating trust. This is not always the case, though. You can imagine me including my own public key in this book and asking you to use it to send me an encrypted message at some email address. Who says my copyeditor did not replace the public key with hers?

Same for Alice: how does she figure out if the public key she received truly is Bob's public key? It's possible that someone in the middle could have tampered with the first message. As you will see in this chapter, cryptography has no real answer to this issue of trust. Instead, it provides different solutions to help in different scenarios. The reason why there is no true solution is that we are trying to bridge reality (real human beings) with a theoretical cryptographic protocol.

> *This whole business of protecting public keys from tampering is the single most difficult problem in practical public key applications. It is the 'Achilles heel' of public key cryptography, and a lot of software complexity is tied up in solving this one problem.*
>
> —Zimmermann et al. ("PGP User's Guide Volume I: Essential Topics," 1992)

Going back to our simple setup where Alice wants to send a file to Bob, and assuming that their untrusted connection is all they have, they have somewhat of an impossible trust issue at hand. Alice has no good way of knowing for sure what truly is Bob's public key. It's a chicken-and-egg type of scenario. Yet, let me point out that if no malicious *active* MITM attacker replaces Bob's public key in the first message, then the protocol is safe. Even if the messages are being passively recorded, it is too late for an attacker to come after the fact to decrypt the second message.

Of course, relying on the fact that your chances of being actively MITM'd are *not too high* is not the best way to undertake cryptography. We, unfortunately, often do not have a way to avoid this. For example, Google Chrome ships with a set of certificate authorities (CAs) that it chooses to trust, but how did you obtain Chrome in the first place? Perhaps you used the default browser of your operating system, which relies on its own set of CAs. But where did that come from? From the laptop you bought. But where did this laptop come from? As you can quickly see, it's "turtles all the way down." At some point, you will have to trust that something was done right.

A threat model typically chooses to stop addressing issues after a specific turtle and considers that any turtle further down is out-of-scope. This is why the rest of the chapter will assume that you have a secure way to obtain some *root of trust*. All systems based on cryptography work by relying on a root of trust, something that a protocol can build security on top of. A root of trust can be a secret or a public value that we start the protocol with or an out-of-band channel that we can use to obtain them.

10.3 The failure of encrypted email

Email was created as (and is still today) an *unencrypted* protocol. We can only blame a time where security was second thought. Email encryption started to become more than just an idea after the release of a tool called *Pretty Good Privacy* (PGP) in 1991. At the time, the creator of PGP, Phil Zimmermann, decided to release PGP in reaction to a bill that almost became law earlier in the same year. The bill would have allowed the US government to obtain all voice and text communications from any electronic communication company and manufacturer. In his 1994 essay "Why Do You Need PGP?", Philip Zimmermann ends with "PGP empowers people to take their privacy into their own hands. There's a growing social need for it. That's why I wrote it."

The protocol was finally standardized in RFC 2440 as *OpenPGP* in 1998 and caught traction with the release of the open source implementation, *GNU Privacy Guard* (GPG), around the same time. Today, GPG is still the main implementation, and people interchangeably use the terms GPG and PGP to pretty much mean the same thing.

10.3.1 PGP or GPG? And how does it work?

PGP, or OpenPGP, works by simply making use of hybrid encryption (covered in chapter 6). The details are in RFC 4880, the last version of OpenPGP, and can be simplified to the following steps:

1. The sender creates an email. At this point the email's content is compressed before it is encrypted.
2. The OpenPGP implementation generates a random symmetric key and symmetrically encrypts the email using the symmetric key.
3. The symmetric key is asymmetrically encrypted to each recipient's public key (using the techniques you learned in chapter 6).
4. All of the intended recipients' encrypted versions of the symmetric key are concatenated with the encrypted message. The email body is replaced with this blob of data and sent to all recipients.
5. To decrypt an email, a recipient uses their private key to decrypt the symmetric key, then decrypts the content of the email using the decrypted symmetric key.

Note that OpenPGP also defines how an email can be signed in order to authenticate the sender. To do this, the plaintext email's body is hashed and then signed using the sender's private key. The signature is then added to the message before being encrypted in step 2. Finally, so that the recipient can figure out what public key to use to verify the signature, the sender's public key is sent along the encrypted email in step 4. I illustrate the PGP flow in figure 10.2.

Exercise

Do you know why the email's content is compressed before it is encrypted and not after?

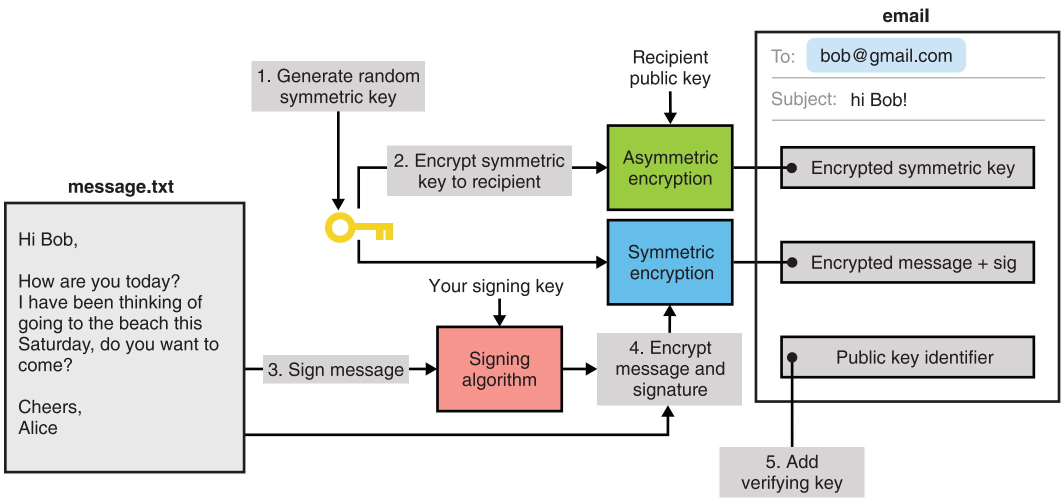


Figure 10.2 PGP's goal is to encrypt and sign messages. When integrated with email clients it does not care about hiding the subject or other metadata.

There's nothing inherently wrong with this design at first sight. It seems to prevent MITM attackers from seeing your email's content, although the subject and other email headers are not encrypted.

> **NOTE** It is important to note that cryptography cannot always hide all metadata. In privacy-conscious applications, metadata is a big problem and can, in the worst cases, de-anonymize you! For example, in end-to-end encrypted protocols, you might not be able to decrypt messages between users, but you can probably tell what their IP addresses are, what the length of the messages they send and receive is, who they commonly talk to (their social graphs), and so on. A lot of engineering is put into hiding this type of metadata.

Yet, in the details, PGP is actually quite bad. The OpenPGP standard and its main implementation, GPG, make use of old algorithms, and backward compatibility prevents them from improving the situation. The most critical issue is that encryption is not authenticated, which means that anyone intercepting an email that hasn't been signed might be able to tamper with the encrypted content to some degree, depending on the exact encryption algorithm used. For this reason alone, I would not recommend anyone to use PGP today.

A surprising flaw of PGP comes from the fact that the signing and encryption operations are composed without care. In 2001, Don Davis pointed out that because of this naive composition of cryptographic algorithms, one can re-encrypt a signed email they received and send that to another recipient. This effectively allows Bob to send you the email Alice sent him as if you were the intended recipient!

If you're wondering, signing the ciphertext instead of the plaintext is still flawed as one could then simply remove the signature that comes with the ciphertext and add their own signature instead. In effect, Bob could pretend that he sent you an email that was actually coming from Alice. I recapitulate these two signing issues in figure 10.3.

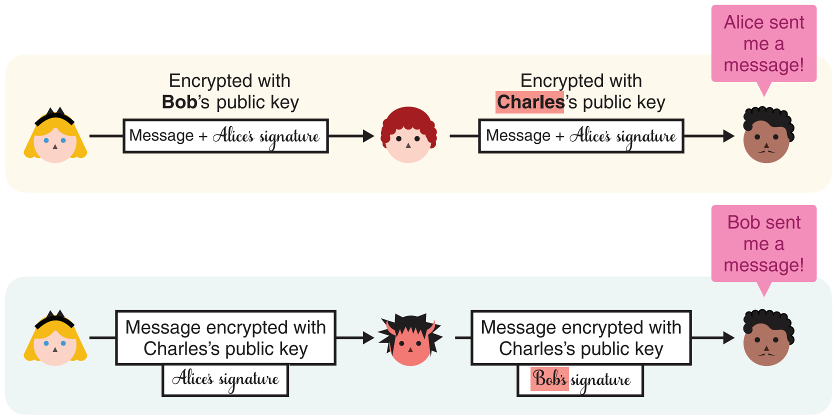

Alice sent me a message!
Encrypted with **Bob**'s public key
Message + Alice's signature
Encrypted with **Charles**'s public key
Message + Alice's signature
Bob sent me a message!
Message encrypted with Charles's public key
Alice's signature
Message encrypted with Charles's public key
Bob's signature

Figure 10.3 In the top diagram, Alice encrypts a message and signature over the message with Bob's public key. Bob can re-encrypt this message to Charles, who might believe that it was intended for him to begin with. This is the PGP flow. In the bottom diagram, this time Alice encrypts a message to Charles. She also signs the encrypted message instead of the plaintext content. Bob, who intercepts the encrypted message, can replace the signature with his own, fooling Charles into thinking that he wrote the content of the message.

Exercise

Can you think of an unambiguous way of signing a message?

The icing on the cake is that the algorithm does not provide *forward secrecy* by default. As a reminder, without forward secrecy, a compromise of your private key implies that all previous emails sent to you encrypted under that key can now be decrypted. You can still force forward secrecy by changing your PGP key, but this process is not straightforward (you can, for example, sign your new key with your older key) and most users just don't bother. To recap, remember that

- PGP uses old cryptographic algorithms.
- PGP does not have authenticated encryption and is, thus, not secure if used without signatures.
- Due to bad design, receiving a signed message doesn't necessarily mean we were the intended recipient.
- There is no forward secrecy by default.

10.3.2 Scaling trust between users with the web of trust

So why am I really talking about PGP here? Well, there is something interesting about PGP that I haven't talked about yet: how do you obtain and how can you trust other people's public keys? The answer is that in PGP, you build trust yourself!

OK, what does that mean? Imagine that you install GPG and decide that you want to encrypt some messages to your friends. To start, you must first find a secure way to obtain your friends' PGP public keys. Meeting them in real life is one sure way to do this. You meet, you copy their public keys on a piece of paper, and then you type those keys back into your laptop at home. Now, you can send your friends signed and encrypted messages with OpenPGP. But this is tedious. Do you have to do this for every person you want to email? Of course not. Let's take the following scenario:

- You have obtained Bob's public key in real life and, thus, you trust it.
- You do not have Mark's public key, but Bob does and he trusts it.

Take a moment here to think about what you could be doing to trust Mark's public key. Bob can simply sign Mark's key, showing you that he trusts the association between the public key and Mark's email. If you trust Bob, you can now trust Mark's public key and add it to your repertoire. This is the main idea behind PGP's concept of *decentralized* trust. It is called the *web of trust* (WOT) as figure 10.4 illustrates.

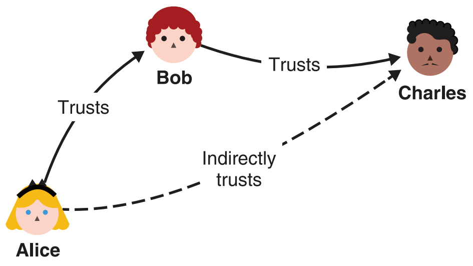


Figure 10.4 The web of trust (WOT) is the concept that users can transitively trust other users by relying on signatures. In this figure, we can see that Alice trusts Bob who trusts Charles. Alice can use Bob's signature over Charles's identity and public key to trust Charles as well.

You will sometimes see "key parties" at conferences, where people meet in real life and sign their respective public keys. But most of that is role-playing, and, in practice, few people rely on the WOT to enlarge their PGP circle.

10.3.3 Key discovery is a real issue

PGP did try another way to solve the issue of discovering public keys—*key registries.* The concept is pretty simple: publish your PGP public key and associated signatures from others that attest to your identity on some public list so that people can find it. In practice, this doesn't work as anyone can publish a key and associated signature purportedly matching your email. In fact, some attackers intentionally spoofed keys on key servers, although possibly more to cause havoc than to spy on emails. In some settings, we can relax our threat model and allow for a trusted authority to attest to identities and public keys. Think of a company managing their employees' emails, for example.

In 1995, the RSA company proposed *Secure/Multipurpose Internet Mail Extensions* (S/MIME) as an extension to the MIME format (which itself is an extension to the email standard) and as an alternative to PGP. S/MIME, standardized in RFC 5751, took an interesting departure from the WOT by using a public key infrastructure to build trust. That is pretty much the only conceptual difference that S/MIME has with PGP. As companies have processes in place to onboard and offboard employees, it makes sense for them to start using protocols like S/MIME in order to bootstrap trust in their internal email ecosystem.

It is important to note that both PGP and S/MIME are generally used over the *Simple Mail Transfer Protocol* (SMTP), which is the protocol used today for sending and receiving emails. PGP and S/MIME were also invented later, and for this reason, their integration with SMTP and email clients is far from perfect. For example, only the body of an email is encrypted not the subject or any of the other email headers. S/MIME, like PGP, is also quite an old protocol that uses outdated cryptography and practices. Like PGP, it does not offer authenticated encryption.

Recent research (Efail: "Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels") on the integration of both protocols in email clients showed that most of them were vulnerable to *exfiltration attacks*, where an attacker who observes encrypted emails can retrieve the content by sending tampered versions to the recipients.

In the end, these shortcomings might not even matter, as most emails being sent and received in the world move along the global network unencrypted. PGP has proven to be quite difficult to use for nontechnical, as well as advanced, users who are required to understand the many subtleties and flows of PGP in order to encrypt their emails. For example, it's not uncommon to see users responding to an encrypted email without using encryption, quoting the whole thread in cleartext. On top of that, the poor (or nonexistent) support for PGP by popular email clients hasn't helped.

> *In the 1990s, I was excited about the future, and I dreamed of a world where everyone would install GPG. Now I'm still excited about the future, but I dream of a world where I can uninstall it.*
>
> —Moxie Marlinspike ("GPG and Me," 2015)

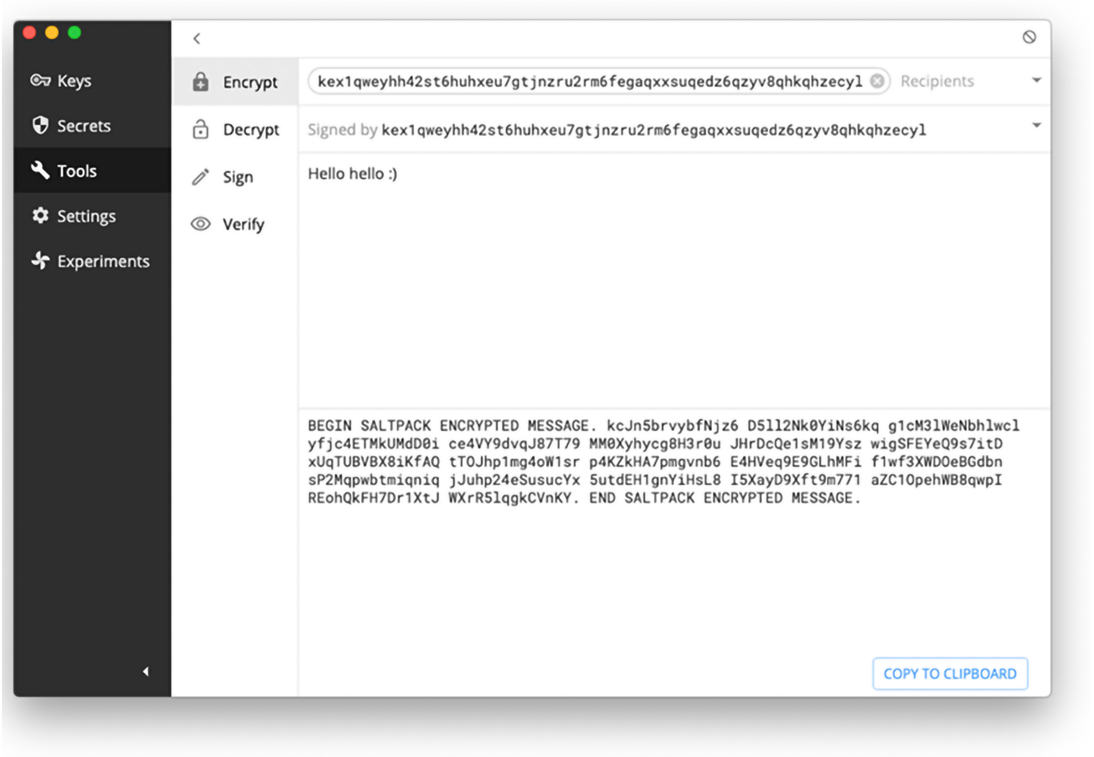
For these reasons, PGP has slowly been losing support (for example, Golang removed support for PGP from its standard library in 2019), while more and more real-world cryptography applications are aiming at replacing PGP and solving its usability problems. Today, it is hard to argue that email encryption will ever have the same level of success and adoption that, for example, HTTPS has.

> *If messages can be sent in plaintext, they will be sent in plaintext. Email is end-to-end unencrypted by default. The foundations of electronic mail are plaintext. All mainstream email software expects plaintext. In meaningful ways, the Internet email system is simply designed not to be encrypted.*
>
> —Thomas Ptacek ("Stop Using Encrypted Email," 2020)

10.3.4 If not PGP, then what?

I spent several pages talking about how a simple design like PGP can fail in a lot of different and surprising ways in practice. Yes, I would recommend against using PGP. While email encryption is still an unsolved problem, alternatives are being developed to replace different PGP use cases.

saltpack is a similar protocol and message format to PGP. It attempts to fix some of the PGP flaws I've talked about. In 2021, saltpack's main implementations are keybase (https://keybase.io) and keys.pub (https://keys.pub). Figure 10.5 illustrates the keys.pub tool.



kex1qweyhh42st6huhxeu7gtjnzru2rm6fegaqxxsuqedz6qzyv8qhkqhzecyl

Signed by kex1qweyhh42st6huhxeu7gtjnzru2rm6fegaqxxsuqedz6qzyv8qhkqhzecyl

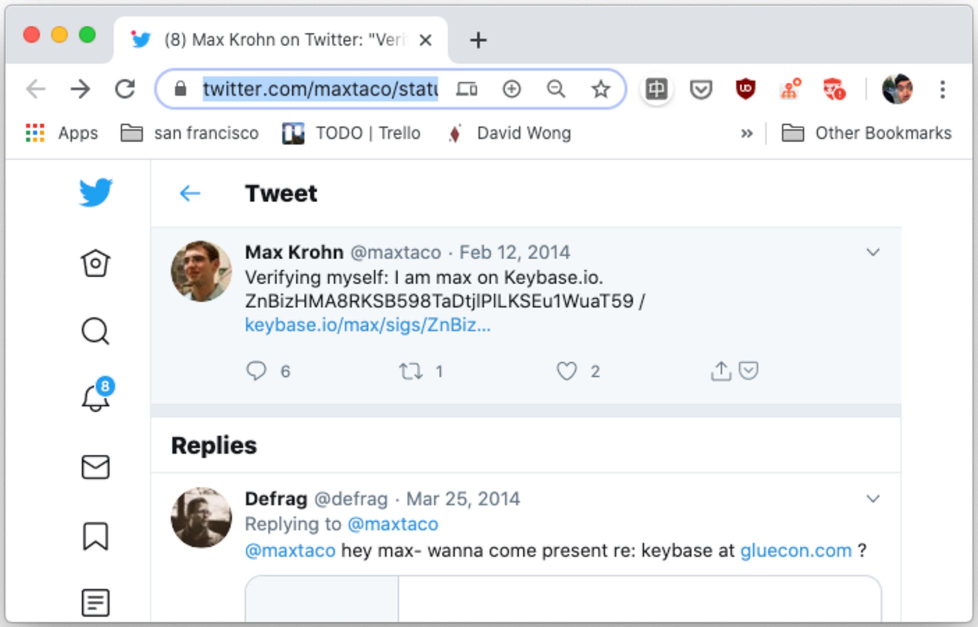
Hello hello :)

```
BEGIN SALTPACK ENCRYPTED MESSAGE. kcJn5brvybfNjz6 D5ll2Nk0YiNs6kq g1cM3lWeNbhlwcl
yfjc4ETMkUMdD0i ce4VY9dvqJ87T79 MM0Xyhycg8H3r0u JHrDcQe1sM19Ysz wigSFEYeQ9s7itD
xUqTUBVBX8iKfAQ tTOJhp1mg4oW1sr p4KZkHA7pmgvnb6 E4HVeq9E9GLhMFi f1wf3XWDOeBGdbn
sP2Mqpwbtmiqniq jJuhp24eSusucYx 5utdEH1gnYiHsL8 I5XayD9Xft9m771 aZC1OpehWB8qwpI
REohQkFH7Dr1XtJ WXrR5lqgkCVnKY. END SALTPACK ENCRYPTED MESSAGE.
```

Figure 10.5 keys.pub is a native desktop application that implements the saltpack protocol. You can use it to import other people's public keys and to encrypt and sign messages to them.

These implementations have all moved away from WOT and allow users to broadcast their public keys on different social networks in order to instill their identity into their public keys (as figure 10.6 illustrates). PGP could obviously not have anticipated this key discovery mechanism as it predates the boom of social networks.

On the other hand, most secure communication nowadays is far from a one-time message, and the use of these tools is less and less relevant. In the next section, I talk



Max Krohn @maxtaco · Feb 12, 2014
Verifying myself: I am max on Keybase.io. ZnBizHMA8RKSB598TaDtjlPILKSEu1WuaT59 / keybase.io/max/sigs/ZnBiz...

6 1 2

Defrag @defrag · Mar 25, 2014
Replying to @maxtaco
@maxtaco hey max- wanna come present re: keybase at gluecon.com ?

Figure 10.6 A keybase user broadcasting their public key on the Twitter social network. This allows other users to obtain additional proof that his identity is linked to a specific public key.

about *secure messaging*, one of the fields that aims to replace the communication aspect of PGP.

10.4 Secure messaging: A modern look at end-to-end encryption with Signal

In 2004, *Off-The-Record* (OTR) was introduced in a white paper titled "Off-the-Record Communication, or, Why Not To Use PGP." Unlike PGP or S/MIME, OTR is not used to encrypt emails but, instead, chat messages; specifically, it extends a chat protocol called the *Extensible Messaging and Presence Protocol* (XMPP).

One of the distinctive features of OTR was *deniability*—a claim that recipients of your messages and passive observers cannot use messages you sent them in a court of justice. Because messages you send are authenticated and encrypted symmetrically with a key your recipient shares with you, they could have easily forged these messages themselves. By contrast, with PGP, messages are signed and are, thus, the inverse of deniable—messages are *non-repudiable*. To my knowledge, none of these properties have actually been tested in court.

In 2010, the Signal mobile phone application (then called TextSecure) was released, making use of a newly created protocol called the *Signal protocol*. At the time, most secure communication protocols like PGP, S/MIME, and OTR were based on *federated protocols*, where no central entity was required for the network to work. The Signal mobile application largely departed from tradition by running a central service and offering a single official Signal client application.

While Signal prevents interoperability with other servers, the Signal protocol is open standard and has been adopted by many other messaging applications including Google Allo (now defunct), WhatsApp, Facebook Messenger, Skype, and many others. The Signal protocol is truly a success story, transparently being used by billions of people including journalists, targets of government surveillance, and even my 92-year-old grandmother (I swear I did not make her install it).

It is interesting to look at how Signal works because it attempts to fix many of the flaws that I previously mentioned with PGP. In this section, I will go over each one of the following interesting features of Signal:

- How can we do better than the WOT? Is there a way to upgrade the existing social graphs with end-to-end encryption? Signal's answer is to use a *trust on first use* (TOFU) approach. TOFU allows users to blindly trust other users the first time they communicate, relying on this first, insecure exchange to establish a long-lasting secure communication channel. Users are then free to check if the first exchange was MITM'd by matching their session secret out of band and at any point in the future.
- How can we upgrade PGP to obtain forward secrecy every time we start a conversation with someone? The first part of the Signal protocol is like most secure transport protocols: it's a key exchange, but a particular one called *Extended Triple Diffie-Hellman* (X3DH). More on that later.
- How can we upgrade PGP to obtain forward secrecy for every single message? This is important because conversations between users can span years, and a compromise at some point in time should not reveal years of communication. Signal addresses this with something called a *symmetric ratchet*.
- What if two users' session secrets are compromised at some point in time? Is that game over? Can we also recover from that? Signal introduces a new security property called *post-compromise security* (PCS) and addresses this with what is called a *Diffie-Hellman* (DH) *ratchet*.

Let's get started! First, we'll see how Signal's flavor of TOFU works.

10.4.1 More user-friendly than the WOT: Trust but verify

One of email encryption's biggest failures was its reliance on PGP and the WOT model to transform social graphs into *secure* social graphs. PGP's original design intended for people to meet in person to perform a *key-signing ceremony* (also called a *key-signing party*) to confirm one another's keys, but this was cumbersome and inconvenient in many and various ways. It is really rare today to see people signing each other's PGP keys.

The way most people use applications like PGP, OTR, Signal, and so on, is to blindly trust a key the first time they see it and to reject any future changes (as figure 10.7 illustrates). This way, only the first connection can be attacked (and this only by an active MITM attacker).

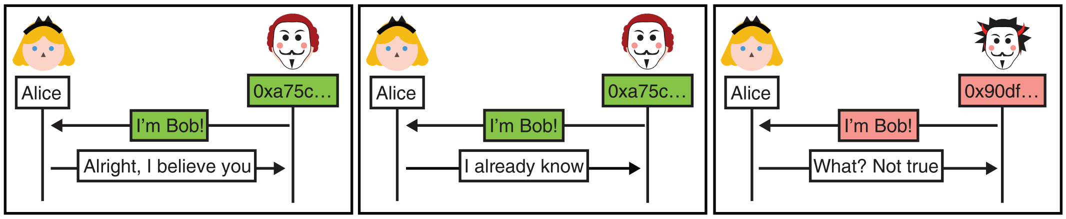
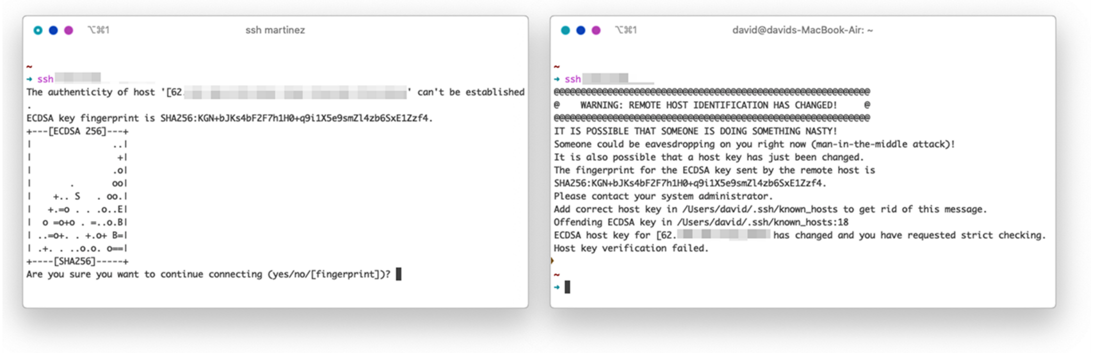


Figure 10.7 Trust on first use (TOFU) allows Alice to trust her first connection but not subsequent connections if they don't exhibit the same public key. TOFU is an easy mechanism to build trust when the chances that the first connection is actively MITM'd are low. The association between a public key and the identity (here Bob) can also be verified after the fact in a different channel.

While TOFU is not the best security model, it is often the best we have and has proven extremely useful. The Secure Shell (SSH) protocol, for example, is often used by trusting the server's public key during the initial connection (see figure 10.8) and by rejecting any future change.



```
~
→ ssh [redacted]
The authenticity of host '[62.[redacted]' can't be established
.
ECDSA key fingerprint is SHA256:KGN+bJKs4bF2F7h1H0+q9i1X5e9smZl4zb6SxE1Zzf4.
+---[ECDSA 256]---+
|               ..|
|                +|
|               .o|
|       .       oo|
|    +.. S   . oo.|
|   +.=o . . .o..E|
|  o =o+o . =..o.B|
| ..=o+. . +.o+ B=|
| .+. . ..o.o. o==|
+----[SHA256]-----+
Are you sure you want to continue connecting (yes/no/[fingerprint])? █
```

```
~
→ ssh [redacted]
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the ECDSA key sent by the remote host is
SHA256:KGN+bJKs4bF2F7h1H0+q9i1X5e9smZl4zb6SxE1Zzf4.
Please contact your system administrator.
Add correct host key in /Users/david/.ssh/known_hosts to get rid of this message.
Offending ECDSA key in /Users/david/.ssh/known_hosts:18
ECDSA host key for [62.[redacted] has changed and you have requested strict checking.
Host key verification failed.

~
→ █
```

Figure 10.8 SSH clients use trust on first use. The first time you connect to an SSH server (the left picture), you have the option to blindly trust the association between the SSH server and the public key displayed. If the public key of the SSH server later changes (right picture), your SSH client prevents you from connecting to it.

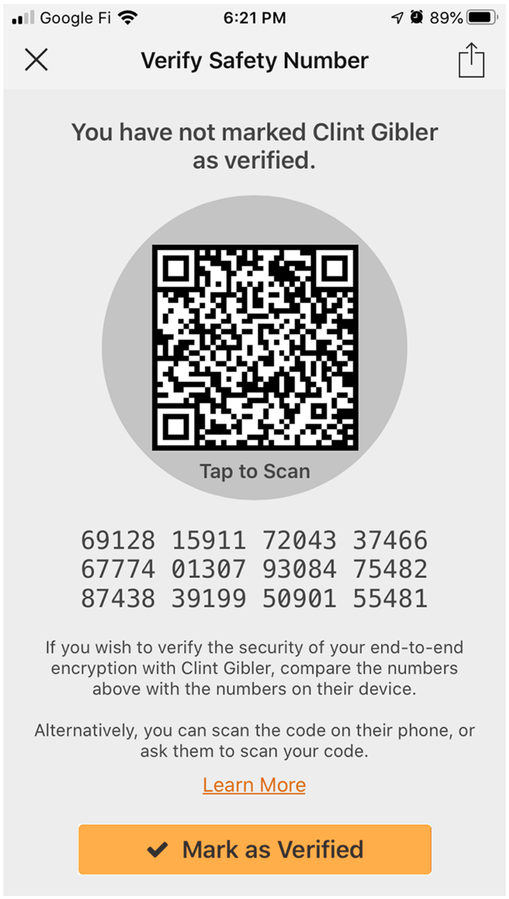
While TOFU systems trust the first key they see, they still allow the user to later verify that the key is, indeed, the right one and to catch any impersonation attempts. In real-world applications, users typically compare *fingerprints*, which are most often hexadecimal representations of public keys or hashes of public keys. This verification is, of course, done out of band. (If the SSH connection is compromised, then the verification is compromised as well.)

NOTE Of course, if users do not verify fingerprints, then they can be MITM'd without knowing it. But that is the kind of tradeoff that real-world applications have to deal with when bringing end-to-end encryption at scale. Indeed, the failure of the WOT shows that security-focused applications must keep usability in mind to get widely adopted.

In the Signal mobile application, a fingerprint between Alice and Bob is computed by:

1. Hashing Alice's identity key prefixed by her username (a phone number in Signal) and interpreting a truncation of that digest as a series of numbers
2. Doing the same for Bob
3. Displaying the concatenation of the two series of numbers to the user

Applications like Signal make use of *QR codes* to let users verify fingerprints more easily as these codes can be lengthy. Figure 10.9 illustrates this use case.



You have not marked Clint Gibler as verified.

```
69128 15911 72043 37466
67774 01307 93084 75482
87438 39199 50901 55481
```

If you wish to verify the security of your end-to-end encryption with Clint Gibler, compare the numbers above with the numbers on their device.

Alternatively, you can scan the code on their phone, or ask them to scan your code.

Figure 10.9 With Signal, you can verify the authenticity and confidentiality of the connection you have with a friend by using a different channel (just like in real life) to make sure the two fingerprints (Signal calls them *safety numbers*) of you and your friend match. This can be done more easily via the use of a QR code, which encodes this information in a scannable format. Signal also hashes the session secret instead of the two users' public keys, allowing them to verify one large string instead of two.

Next, let's see how the Signal protocol works under the hood—specifically, how Signal manages to be forward secure.

10.4.2 X3DH: the Signal protocol's handshake

Most secure messaging apps before Signal were *synchronous*. This meant that, for example, Alice wasn't able to start (or continue) an end-to-end encrypted conversation with Bob if Bob was not online. The Signal protocol, on the other hand, is *asynchronous* (like email), meaning that Alice can start (and continue) a conversation with people that are offline.

Remember that *forward secrecy* (covered in chapter 9) means that a compromise of keys does not compromise previous sessions and that forward secrecy usually means that the key exchanges are interactive because both sides have to generate ephemeral Diffie-Hellman (DH) key pairs. In this section, you will see how Signal uses *non-interactive* key exchanges (key exchanges where one side is potentially offline) that are still forward secure. OK, let's get going.

To start a conversation with Bob, Alice initiates a key exchange with him. Signal's key exchange, X3DH, combines three (or more) DH key exchanges into one. But before you learn how that works, you need to understand the three different types of DH keys that Signal uses:

- *Identity keys*—These are the long-term keys that represent the users. You can imagine that if Signal only used identity keys, then the scheme would be fairly similar to PGP, and there would be no forward secrecy.
- *One-time prekeys*—In order to add forward secrecy to the key exchange, even when the recipient of a new conversation is not online, Signal has users upload multiple *single-use* public keys. They are simply ephemeral keys that are uploaded in advance and are deleted after being used.
- *Signed prekeys*—We could stop here, but there's one edge case missing. Because the one-time prekeys that a user uploads can, at some point, be depleted, users also have to upload a *medium-term* public key that they sign: a signed prekey. This way, if no more one-time prekeys are available on the server under your username, someone can still use your signed prekey to add forward secrecy up to the last time you changed your signed prekey. This also means that you have to periodically rotate your signed prekey (for example, every week).

This is enough to preview what the flow of a conversation creation in Signal looks like. Figure 10.10 presents an overview.

Let's go over each of these steps in more depth. First, a user registers by sending the following:

- One identity key
- One signed prekey and its signature
- A defined number of one-time prekeys

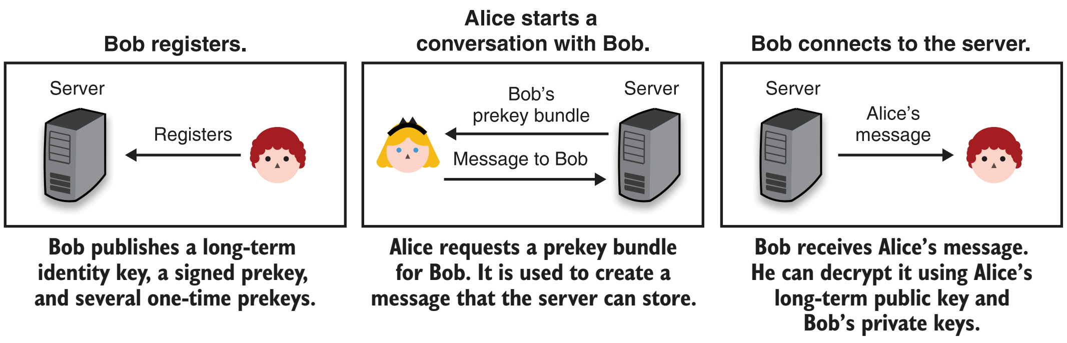

Bob registers.
Server
Registers
Bob publishes a long-term identity key, a signed prekey, and several one-time prekeys.
Alice starts a conversation with Bob.
Bob's prekey bundle
Server
Message to Bob
Alice requests a prekey bundle for Bob. It is used to create a message that the server can store.
Bob connects to the server.
Server
Alice's message
Bob receives Alice's message. He can decrypt it using Alice's long-term public key and Bob's private keys.

Figure 10.10 Signal's flow starts with a user registering with a number of public keys. If Alice wants to talk to Bob, she first retrieves Bob's public keys (called a *prekey bundle*), then she performs an X3DH key exchange with these keys and creates an initial message using the output of the key exchange. After receipt of the message, Bob can perform the same on his side to initialize and continue the conversation.

At this point, it is the responsibility of the user to periodically rotate the signed prekey and upload new one-time prekeys. I recap this flow in figure 10.11.

NOTE Signal makes use of the identity key to perform signatures over signed prekeys and key exchanges during the X3DH key exchange. While I've warned against using the same key for different purposes, Signal has deliberately analyzed that, in their case, there should be no issue. This does not mean that this would work in *your* case and with *your* key exchange algorithm. I would advise against using a key for different purposes in general.

After the step introduced in figure 10.11, Alice (going back to our example) would then start a conversation with Bob by retrieving:

- Bob's identity key.
- Bob's current signed prekey and its associated signature.
- If there are still some, one of Bob's one-time prekeys (the server then deletes the one-time prekey sent to Alice).

Alice can verify that the signature over the signed prekey is correct. She then performs the X3DH handshake with:

- All of the public keys from Bob
- An ephemeral key pair that she generates for the occasion in order to add forward secrecy
- Her own identity key

The output of X3DH is then used in a post-X3DH protocol, which is used to encrypt her messages to Bob (more on that in the next section). X3DH is composed of three

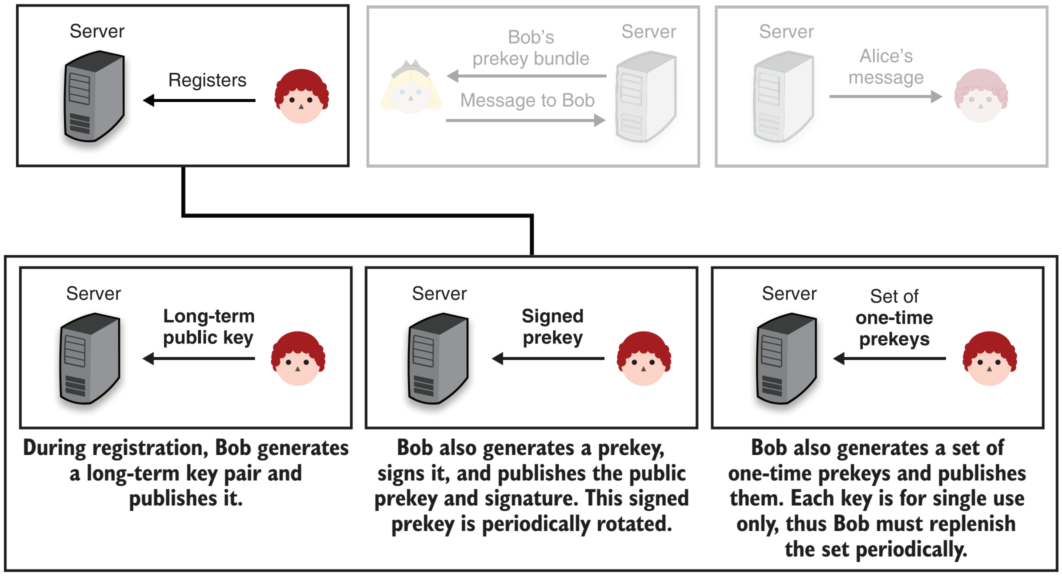

Server
Registers
Bob's prekey bundle
Server
Message to Bob
Server
Alice's message
Server
Long-term public key
During registration, Bob generates a long-term key pair and publishes it.
Server
Signed prekey
Bob also generates a prekey, signs it, and publishes the public prekey and signature. This signed prekey is periodically rotated.
Server
Set of one-time prekeys
Bob also generates a set of one-time prekeys and publishes them. Each key is for single use only, thus Bob must replenish the set periodically.

Figure 10.11 Building on figure 10.10, the first step is for a user to register by generating a number of DH key pairs and sending the public parts to a central server.

(optionally, four) DH key exchanges, grouped into one. The DH key exchanges are between:

1. The identity key of Alice and the signed prekey of Bob
2. The ephemeral key of Alice and the identity key of Bob
3. The ephemeral key of Alice and the signed prekey of Bob
4. If Bob still has a one-time prekey available, his one-time prekey and the ephemeral key of Alice

The output of X3DH is the concatenation of all of these DH key exchanges, passed to a key derivation function (KDF), which we covered in chapter 8. Different key exchanges provide different properties. The first and second ones are here for mutual authentication, while the last two are here for forward secrecy. All of this is analyzed in more depth in the X3DH specification (https://signal.org/docs/specifications/x3dh/), which I encourage you to read if you want to know more as it is well written. Figure 10.12 recaps this flow.

Alice now can send Bob her identity public key, the ephemeral public key she generated to start the conversation, and other relevant information (like which of Bob's one-time prekeys she used). Bob receives the message and can perform the exact same X3DH key exchange with the public keys contained in it. (For this reason, I skip illustrating the last step of this flow.) If Alice used one of Bob's one-time prekeys, Bob gets rid of it. What happens after X3DH is done? Let's look at that next.

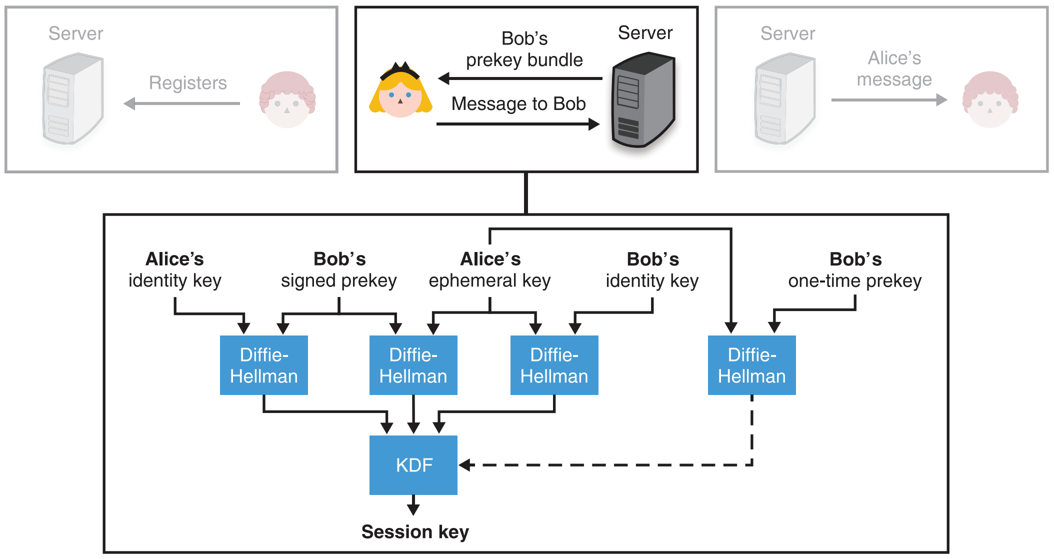


Figure 10.12 Building on figure 10.10, to send a message to Bob, Alice fetches a prekey bundle containing Bob's long-term key, Bob's signed prekey, and optionally, one of Bob's one-time prekeys. After performing different key exchanges with the different keys, all outputs are concatenated and passed into a KDF to produce an output used in a subsequent post-X3DH protocol to encrypt messages to Bob.

10.4.3 Double Ratchet: Signal's post-handshake protocol

The post-X3DH phase lives as long as the two users do not delete their conversations or lose any of their keys. For this reason, and because Signal was designed with SMS conversations in mind, where the time between two messages might be counted in months, Signal introduces *forward secrecy* at the message level. In this section, you will learn how this post-handshake protocol (called the *Double Ratchet*) works.

But first, imagine a simple post-X3DH protocol. Alice and Bob could have taken the output of X3DH as a session key and use it to encrypt messages between them as figure 10.13 illustrates.

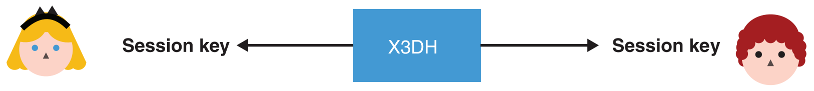


Figure 10.13 Naively, a post-X3DH protocol could simply use the output of X3DH as a session key to encrypt messages between Alice and Bob.

We usually want to separate the keys used for different purposes though. What we can do is to use the output of X3DH as a *seed* (or *root key*, according to the Double Ratchet specification) to a KDF in order to derive two other keys. Alice can use one key to encrypt messages to Bob, and Bob can use the other key to encrypt messages to Alice. I illustrate this in figure 10.14.

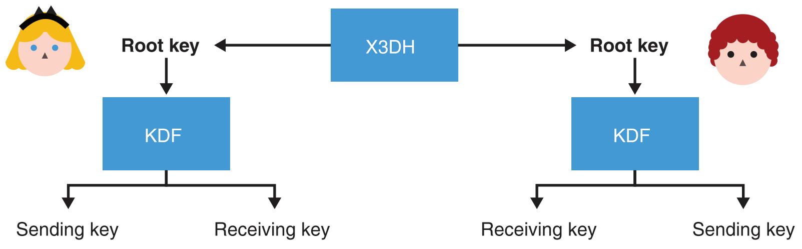


Figure 10.14 Building on figure 10.13, a better post-X3DH protocol would make use of a KDF with the output of the key exchanges to differentiate keys used to encrypt Bob's and Alice's messages. Here Alice's sending key is the same as Bob's receiving key, and Bob's sending key is the same as Alice's receiving key.

This approach could be enough, but Signal notes that texting sessions can last for years. This is unlike the TLS sessions of chapter 9 that are usually expected to be short-lived. Because of this, if at any point in time the session key is stolen, all previously recorded messages can be decrypted!

To fix this, Signal introduced what is called a *symmetric ratchet* (as figure 10.15 illustrates). The *sending key* is now renamed a *sending chain key* and is not used directly to

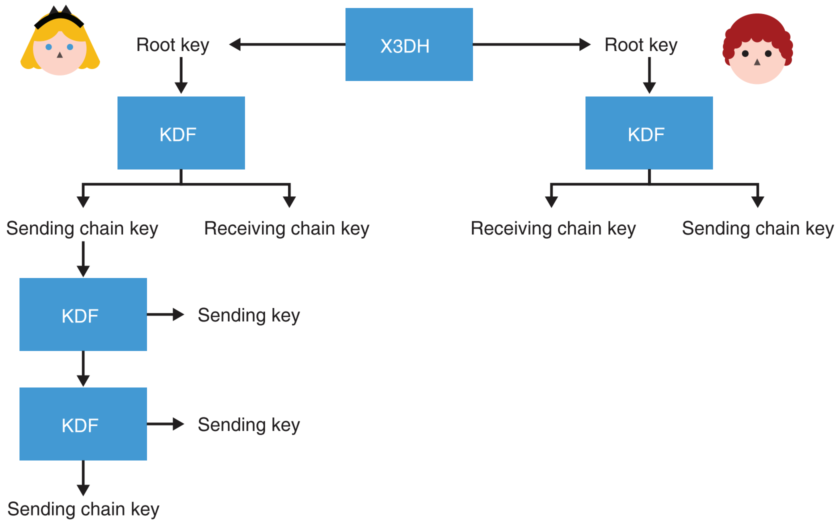


Figure 10.15 Building on figure 10.14, forward secrecy can be introduced in the post-X3DH protocol by *ratcheting* (passing into a KDF) a chain key every time one needs to send a message, and ratcheting another chain key every time one receives a message. Thus, the compromise of a sending or receiving chain key does not allow an attacker to recover previous ones.

encrypt messages. When sending a message, Alice continuously passes that sending chain key into a one-way function that produces the next sending chain key as well as the actual sending keys to encrypt her messages. Bob, on the other hand, will have to do the same but with the receiving chain key. Thus, by compromising one sending key or sending chain key, an attacker cannot recover previous keys. (And the same is true when receiving messages.)

Good. We now have forward secrecy baked into our protocol and at the message level. Every message sent and received protects all previously sent and received messages. Note that this is somewhat debatable as an attacker who compromises a key probably does this by compromising a user's phone, which will likely contain all previous messages in cleartext next to the key. Nevertheless, if both users in a conversation decide to delete previous messages (for example, by using Signal's "disappearing messages" feature), the forward secrecy property is achieved.

The Signal protocol has one last interesting thing I want to talk about: PCS (*post-compromise security*, also called *backward secrecy* as you learned in chapter 8). PCS is the idea that if your keys get compromised at some point, you can still manage to recover as the protocol will heal itself. Of course, if the attacker still has access to your device after a compromise, then this is for nothing.

PCS can work only by reintroducing new entropy that a nonpersistent compromise wouldn't have access to. The new entropy has to be the same for both peers. Signal's way of finding such entropy is by doing an ephemeral key exchange. To do this, the Signal protocol continuously performs key exchanges in what is called a *DH ratchet*. Every message sent by the protocol comes with the current ratchet public key as figure 10.16 illustrates.

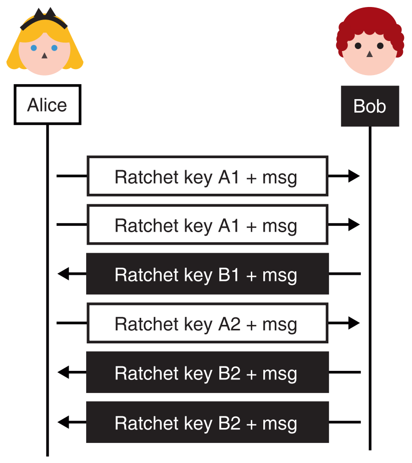


Figure 10.16 The Diffie-Hellman (DH) ratchet works by advertising a ratchet public key in every message sent. This ratchet public key can be the same as the previous one, or it can advertise a new ratchet public key if a participant decides to refresh theirs.

When Bob notices a new ratchet key from Alice, he must perform a new DH key exchange with Alice's new ratchet key and Bob's own ratchet key. The output can then be used with the symmetric ratchet to decrypt the messages received. I illustrate this in figure 10.17.

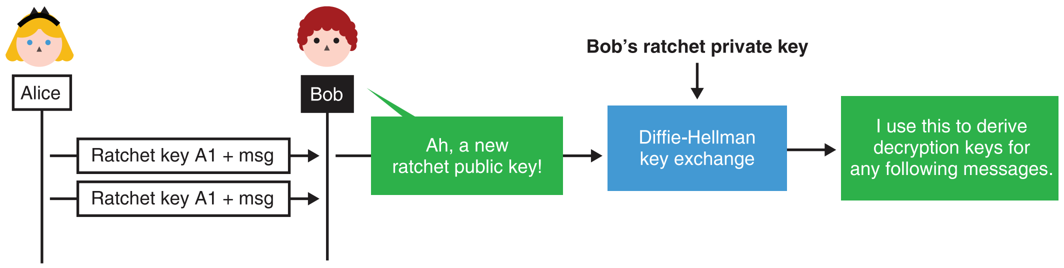

Alice
Bob
Ratchet key A1 + msg
Ratchet key A1 + msg
Ah, a new ratchet public key!
Bob's ratchet private key
Diffie-Hellman key exchange
I use this to derive decryption keys for any following messages.

Figure 10.17 When receiving a new ratchet public key from Alice, Bob must do a key exchange with it and his own ratchet key to derive decryption keys. This is done with the symmetric ratchet. Alice's messages can then be decrypted.

Another thing that Bob must do when receiving a new ratchet key is to generate a new random ratchet key for himself. With his new ratchet key, he can perform another key exchange with Alice's new ratchet key, which he then uses to encrypt messages to her. This should look like figure 10.18.

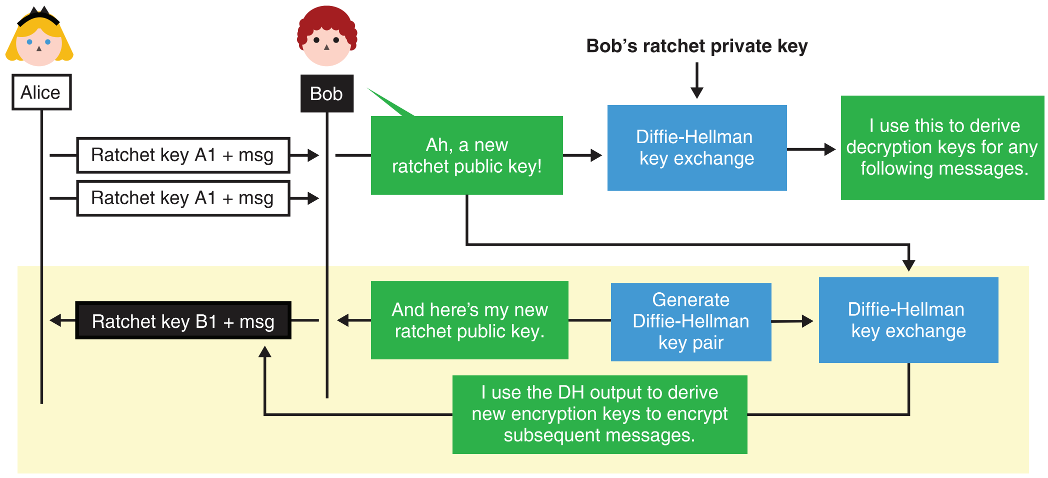

Alice
Bob
Ratchet key A1 + msg
Ratchet key A1 + msg
Ah, a new ratchet public key!
Bob's ratchet private key
Diffie-Hellman key exchange
I use this to derive decryption keys for any following messages.
Ratchet key B1 + msg
And here's my new ratchet public key.
Generate Diffie-Hellman key pair
Diffie-Hellman key exchange
I use the DH output to derive new encryption keys to encrypt subsequent messages.

Figure 10.18 Building on figure 10.17, after receiving a new ratchet key Bob must also generate a new ratchet key for himself. This new ratchet key is used to derive encryption keys and is advertised to Alice in his next series of messages (up until he receives a new ratchet key from Alice).

This back and forth of key exchanges is mentioned as a "ping-pong" in the Double Ratchet specification:

This results in a "ping-pong" behavior as the parties take turns replacing ratchet key pairs. An eavesdropper who briefly compromises one of the parties might learn the value of a current ratchet private key, but that private key will eventually be replaced with an uncompromised one. At that point, the Diffie-Hellman calculation between ratchet key pairs will define a DH output unknown to the attacker.

—The Double Ratchet Algorithm

Finally, the combination of the DH ratchet and the symmetric ratchet is called the *Double Ratchet*. It's a bit dense to visualize as one diagram, but figure 10.19 attempts to do so.

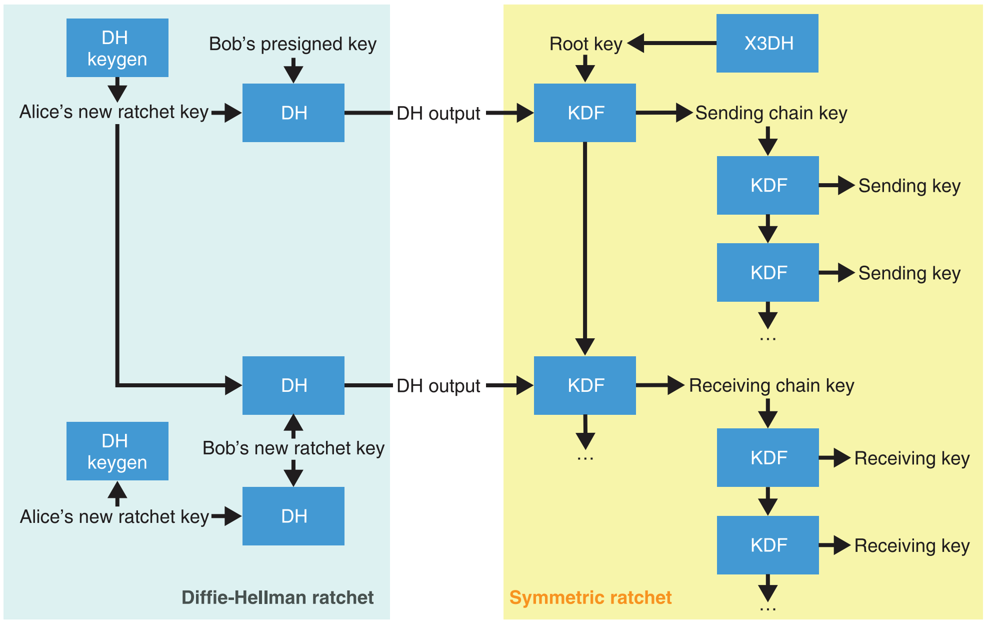


Figure 10.19 The Double Ratchet (from Alice's point of view) combines the DH ratchet (on the left) with the symmetric ratchet (on the right). This provides PCS as well as forward secrecy to the post-X3DH protocol. In the first message, Alice does not yet know Bob's ratchet key so she uses his presigned key instead.

I know this last diagram is quite dense, so I encourage you to take a look at Signal's specifications, which are published on https://signal.org/docs. They provide another well-written explanation of the protocol.

10.5 The state of end-to-end encryption

Today, most secure communications between users happen through secure messaging applications instead of encrypted emails. The Signal protocol has been the clear winner in its category, being adopted by many proprietary applications and also by open source and federated protocols like XMPP (via the OMEMO extension) and Matrix (a modern alternative to IRC). On the other hand, PGP and S/MIME are being dropped as published attacks have led to a loss of trust.

What if you want to write your own end-to-end encrypted messaging app? Unfortunately, a lot of what's being used in this field is ad hoc, and you would have to fill in many of the details yourself in order to obtain a full-featured and secure system. Signal has open sourced a lot of its code, but it lacks documentation and can be hard to use correctly. On the other hand, you might have better luck using a decentralized open source solution like Matrix, which might prove easier to integrate with. This is what the French government has done.

Before we close this chapter, there are also a number of open questions and active research problems that I want to talk about. For example

- Group messaging
- Support for multiple devices
- Better security assurances than TOFU

Let's start with the first item: *group messaging*. At this point, while implemented in different ways by different applications, group messaging is still being actively researched. For example, the Signal application has clients make sense of group chats. Servers only see pairs of users talking—never less, never more. This means that clients have to encrypt a group chat message to all of the group chat participants and send them individually. This is called *client-side fanout* and does not scale super well. It is also not too hard for the server to figure out what are the group members when it sees Alice, for example, sending several messages of the same length to Bob and Charles (figure 10.20).

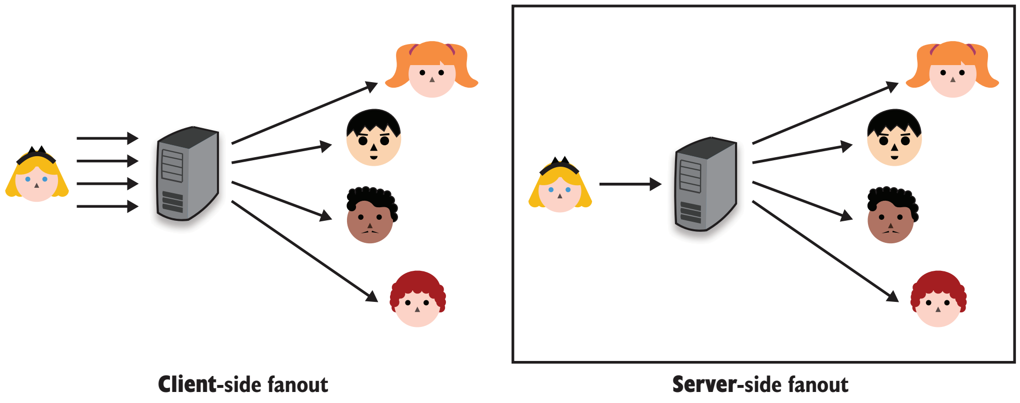


Figure 10.20 There are two ways to approach end-to-end encryption for group chats. A client-side fanout approach means that the client has to individually message each recipient using their already existing encrypted channel. This is a good approach to hide group membership from the server. A server-side fanout approach lets the server forward a message to each group chat participant. This is a good approach to reduce the number of messages sent from the client's perspective.

WhatsApp, on the other hand, uses a variant of the Signal protocol where the server is aware of group chat membership. This change allows a participant to send a single encrypted message to the server that, in turn, will have the responsibility to forward it to the group members. This is called *server-side fanout*.

Another problem of group chat is *scaling* to groups of a large memberset. For this, many players in the industry have recently gathered around a *Messaging Layer Security* (MLS) standard to tackle secure group messaging at scale. But there seems to be a lot of work to be done, and one can wonder, is there really any confidentiality in a group chat with more than a hundred participants?

NOTE This is still an area of active research, and different approaches come with different tradeoffs in security and usability. For example, in 2021, no group chat protocol seems to provide *transcript consistency*, a cryptographic property that ensures that all participants of a group chat see the same messages in the same order.

Support for multiple devices is either not a thing or implemented in various ways, most often by pretending that your different devices are different participants of a group chat. The TOFU model can make handling multiple devices quite complicated because having different identity keys per device can become a real key management problem. Imagine having to verify fingerprints for each of your devices and each of your friends' devices. Matrix, for example, has a user sign their own devices. Other users then can trust all your devices as one entity by verifying their associated signatures.

Finally, I mentioned that the TOFU model is also not the greatest as it is based on trusting a public key the first time we see it, and most users do not verify later that the fingerprints match. Can something be done about this? What if the server decides to impersonate Bob to Alice only? This is a problem that *Key Transparency* is trying to tackle. Key Transparency is a protocol proposed by Google, which is similar to the Certificate Transparency protocol that I talked about in chapter 9. There is also some research making use of the blockchain technology that I'll talk about in chapter 12 on cryptocurrencies.

Summary

- End-to-end encryption is about securing communications among real human beings. A protocol implementing end-to-end encryption is more resilient to vulnerabilities that can happen in servers sitting in between users and can greatly simplify legal requirements for companies.
- End-to-end encryption systems need a way to bootstrap trust between users. This trust can come from a public key that we already know or an out-of-band channel that we trust.
- PGP and S/MIME are the main protocols that are used to encrypt emails today, yet none of them are considered safe to use as they make use of old cryptographic

algorithms and practices. They also have poor integration with email clients that have been shown to be vulnerable to different attacks in practice.

  – PGP uses a web of trust (WOT) model, where users sign each other public keys in order to allow others to trust them.
  – S/MIME uses a public key infrastructure to build trust between participants. It is most commonly used in companies and universities.

- An alternative to PGP is saltpack, which fixes a number of issues while relying on social networks to discover other people's public keys.
- Emails will always have issues with encryption as the protocol was built without encryption in mind. On the other hand, modern messaging protocols and applications are considered better alternatives to encrypted emails as they are built with end-to-end encryption in mind.
  – The Signal protocol is used by most messaging applications to secure end-to-end communications between users. Signal messenger, WhatsApp, Facebook Messenger, and Skype all advertise their use of the Signal protocol to secure messages.
  – Other protocols, like Matrix, attempt to standardize federated protocols for end-to-end encrypted messaging. Federated protocols are open protocols that anyone can interoperate with (as opposed to centralized protocols that are limited to a single application).